ECAN1000 CAN Interface Board Driver for Windows 98/NT4/2000/XP

Version 3.1.x

User's Manual and API Reference

(Real Time Devices)

RTD Embedded Technologies, Inc.

"Accessing the Analog World"®

ISO9001 and AS9100 Certified

SWM-640020005
Rev. F

RTD Embedded Technologies, INC.
103 Innovation Blvd.
State College, PA 16803-0906

Phone: +1-814-234-8087
FAX: +1-814-234-5218

E-mail
sales@rtd.com
techsupport@rtd.com

web site
http://www.rtd.com

Revision History

08/15/2001	Original Release.
12/12/2003	Release 2.1 AllowBufferOverwrite function added. Manual format updated to RTD USA standard format. Getting Started section added. Added Optional Calling Syntax sections. Removed Vxd Parameter from CreateHandle function. Added Queue Size variables to SetupBoard API. Added LoadPortBitDir, Read_Digital_IO, and Write_Digital_IO functions. Updated SetLeds function.
08/31/2004	SetBitRate, SetSingleFilterStandard, SetSingleFilterExtended, SetDualFilterStandard, and SetDualFilterExtended functions added. MessageObjectSetup description added. Added Section on Message Filters. Added Section on Transmission and Reception of Messages
07/24/2006	Removed Windows 98 and Windows NT support. Added Windows Server 2003 support. Added the Can3echo and Can3src examples.
06/16/2009	Renamed the bit-rate '6150' to '6250'. Removed installation instructions as they are now in ReadMe.txt. Added a Table of Contents entry for Ecan_SetBitRate.
04/16/2010	Updated the manual to apply exclusively to the ECAN1000 because the ECAN527 no longer uses this manual.

Ecan1000 Driver for Windows 98/NT4/2000/XP
Published by:

RTD Embedded Technologies, Inc.
103 Innovation Blvd.
State College, PA 16803-0906

TABLE OF CONTENTS

GETTING STARTED

Files in this Distribution:

The files in this distribution are designed for the Ecan1000.

There are six sample programs installed during setup. These are located in the samples directory, which by default is located in: C:\RTD\Ecan1000\Samples. The executables for the samples are installed (by default) into: C:\RTD\Ecan1000\Samples\!Exe.

The Samples are called:

Can1s -- Transmits a single message.
Can1r -- Receives a single message.
Can2s -- Transmits messages with random data until CTRL+C is pressed.
Can2r -- Receives messages from Can2s and displays average BPS.
Test_Ecan -- GUI Application that can send and receive messages. Also detects and deals with any send errors immediately.
Ecan1000 -- Message filtering sample

For Windows 98, 2000 and XP users, the Test_Ecan program allows the user to select an Ecan card and test it by pressing the 'Test Device' Button on the main window. The program also allows the user to test the driver. When the 'Test Driver' button is pressed on the main window, a second window opens which allows transmitting and receiving messages. To test the driver, first select a transmitter or receiver device in the list boxes. Then press the Initialize button. A single message may be sent by pressing 'Send Message'. Multiple messages may be sent by pressing 'Cyclic Send'.

Windows 98 users will see a file called FastInst.exe and Windows 2000,XP users will see a file called DrvInst.exe in the main ECAN1000 directory. These files allow quick creation of Ecan1000 devices in the Hardware devices list. To use them, type in the executable name followed by the fully qualified path and name of the INF File. For example:

```
DrvInstall C:\RTD\Ecan1000\Ecan1000.inf
```

Will add another ECAN1000 card to the list in the Device Manager.

Compiling and Using Samples:

The samples have been written using Microsoft Visual C++ 6.0 but should work with other C++ compilers with little or no modification. Each sample links to the EcanDll.lib file in the LIB directory which by default is located in: C:\RTD\Ecan1000\Lib.

Each sample also includes EcanDLL.h which itself includes EcanIoctl.h , both of which can be found in the Lib directory.

Each API call in this manual is in the EcanDLL.dll dynamic link library and is defined in the EcanDll.h file. To use the API functions, include EcanDLL.h in your project (remember it also includes EcanIoctl.h) and include EcanDLL.lib in your link library list. Then remember to distribute EcanDll.DLL with your application

Using the API with other Computer Languages:

For those people who prefer to use other computer languages (such as Visual Basic) a map file of the DLL has been included. The map file contains the decorated names of the functions that are exported in the DLL. In Visual Basic, use the Declare keyword to define the function you wish to use. For Example: The Declaration for the AllowBufferOverwrite function would be: Private Declare Function AllowBufferOverwrite Lib "ECANDLL" Alias "?Ecan_AllowBufferOverwrite@@YG_NPAX@Z" (ByVal Handle as long) as Boolean

Adding Another Ecan Board

If you add another card to the system you should follow these steps to add a new device to Windows.

For Windows XP:
Go to Add Hardware in the Control Panel. When asked, select 'Yes, I have already connected the hardware'. Then select 'Add a new hardware device' from the list. Then select 'Install the Hardware that I manually select from a list'. Select 'Show all devices'. Wait while the system creates a device list. Press the 'Have Disk…' button. When you are prompted for a location to copy the manufacturer's files from, browse to the directory you installed the Ecan1000 drivers to (usually C:\RTD\ECAN1000). The screen will now list the ECAN1000 CAN Interface board. Press Next to install the new device. When the installation is complete you will have the opportunity to 'View or change resources for this hardware' Select it and configure the IRQ and Base Address of the new board.

For Windows 98:
Go to the Add New Hardware wizard in the Control Panel. Allow Windows to search for new Plug and Play devices. When the wizard displays a list of devices in the system (or a blank list) select 'No, the device isn't in the list'. On the next page select 'No, I want to select the hardware from a list.' Highlight 'Other Devices' from the displayed list of device categories. On the next page press 'Have Disk…' When prompted for a location to copy the manufacturer's files from, browse to the directory you installed the ECAN1000 software in. (Usually C:\RTD\ECAN1000). The screen will now list the ECAN1000 CAN Interface board, Press Next. Continue to proceed through the wizard pages. After the system has rebooted, remember to go to the Device Manager to configure the correct IRQ and Base Address of the new board.

For Windows 2000:
Go to the Add/Remove hardware wizard in the Control Panel. Select 'Add/Troubleshoot a device.' Wait while the system searches for plug and play devices. When it displays a list of devices, select 'Add a new device'. When prompted, select 'No, I want to select the hardware from a list'. Select 'Other Devices' from the list of device categories. When the list of devices is displayed, press 'Have Disk…' When prompted for a location to copy the manufacturers files from, browse to the directory you installed the ECAN1000 software in. (Usually C:\RTD\ECAN1000). The screen will now list the ECAN1000 CAN Interface board, Press

Next. The wizard will now prompt you for the IRQ and Base Address of the new board. When done, proceed through the wizard pages to the final page.

For Windows NT:
Use the supplied Test Ecan software. On the main screen press ‘Add Device’ and enter the IRQ and Base Address of the new board.

MESSAGE RECEPTION AND TRANSMISSION

At the heart of CAN communication is the ability to transmit and receive messages. In this package are two examples which demonstrate message sending and reception. CAN1S.exe is the sending example and CAN1R.exe is the reception example. Both samples are capable of using any supported bitrate.

CAN1S.EXE

The setup for sending messages looks like the following:

First a handle to the card is required. To do this we use the CreateHandle call. The call requires the device number and the card type as a bool.

```
TransmitterHND = Ecan_CreateHandle(0,Transmitter_Type);
```

Next we select the BitRate that we want communication to use. We do it now before we start the card so that we don't have to stop the card and re-start it later. The BitRate value we send to it is one of the 'BitRates' enumeration values.

```
Ecan_SetBitRate(TransmitterHND,BitRate);
```

Next we start the card. It is normally a good idea to check the return code from this function.

```
If (!Ecan_StartBoard(TransmitterHND))
 printf("Could not Start the board!\n");
```

In order to send a message we need to create an ECAN_MESSAGE_STRUCTURE.

```
ECAN_MESSAGE_STRUCTURE Msg;
```

We would like to send a standard message (one that has 11 bits in the ID).

```
Msg.Extended = false;
```

We would like to send the message on the default message channel.

```
Msg.Channel = 0;
```

We want to send the full 8 bytes

```
Msg.DataLength = 8;
```

For this example we fill the data bytes with random data.

```
for (i = 0; i < 8;i++)
 Msg.Data[i] = rand();
```

Finally we transmit the message.

```
Ecan_SendMessaage(TransmitterHND,&Msg);
```

For this simple example we used the default sending object to send the message. The default channel should always be used.

CAN1R.EXE

The setup for receiving messages looks like the following:

First a handle to the card is required. To do this we use the CreateHandle call. The call requires the device number and the card type as a bool.

```
ReceiverHND = Ecan_CreateHandle(0,Receiver_Type);
```

For this example we also want an event to be triggered every time a message is received.

```
o_ptr->hEvent = CreateEvent(0,FALSE,FALSE,NULL);

Ecan_SetupBoard(ReceiverHND,o_ptr->hEvent,o_ptr);
```

Next we select the BitRate that we want communication to use. We do it now before we start the card so that we don't have to stop the card and re-start it later. The BitRate value we send to it is one of the 'BitRates' enumeration values.

```
Ecan_SetBitRate(ReceiverHND,BitRate);
```

Next we start the card. It is normally a good idea to check the return code from this function.

```
If (!Ecan_StartBoard(ReceiverHND))
 printf("Could not Start the board!\n");
```

Next we want to make sure there are no masks set that might filter incoming messages.

```
ECAN_FILTER_STRUCTURE Filter;
Filter.Clear();
Filter.SetAcceptCode(0,false);
Filter.SetAcceptMask(0,false,ReceiverType);
Ecan_SetFilter(ReceiverHND,&Filter);
```

Now we wait for a message to be received. When a message is received the hEvent will be triggered. The first thing we do is get the number of messages waiting. We do this because it is possible that messages have come in so close together that we haven't been fast enough to service them individually.

```
Ecan_GetQueuesCounts(ReceiverHND,NULL,&RX_count);
```

Then, for each message in the queue we call GetInterrupts. This function tells us if there has been a receive interrupt. It also gets a message ready for us to read if there is one available.

```
Interrupt = Ecan_GetInterrupts(ReceiverHND,&QueueCount);
```

Then we check the value of Interrupt to see if there was a receive interrupt or if the queue filled up.

```
If (Interrupt == 0xFF)
 printf("Receive Queue is Full\n");
```

Next if the Interrupt was the receive bit then we get the message.

```
If (Interrupt & Int_RI_Bit)
 Ecan_GetMessage(ReceiverHND,&IncomingMsg);
```

If there were more messages waiting in the queue we go back and GetInterrupts again.

MESSAGE FILTERING

One of the most complex parts of CANbus setup is the implementation of message filters. Ecan1000 cards have fully featured hardware filtering capabilities.

Ecan1000 Message Filtering

For the Ecan1000 there are four different functions that apply to the four different types of filtering that the board can perform. These are:

Ecan_SetSingleFilterStandard

Ecan_SetSingleFilterExtended

Ecan_SetDualFilterStandard

Ecan_SetDualFilterExtended

The functions that end in 'standard' are for standard message frames (those that have an 11 bit ID). The functions that end in 'extended' are for extended messages (messages with a 29 bit ID). Each of these functions is used in the example program called Ecan1000.exe. Simply uncomment the section that uses the filtering method you are interested in and re-compile the program to test the filter. Each filter type is described in detail in Philips's document '*SJA1000 Stand-alone CAN controller DATA SHEET*'. Some of that document will be repeated here for clarity.

All of the filtering methods have two things in common. They all deal with one or more Acceptance Code Registers (ACR) and one or more Acceptance Mask Registers (AMR). An Acceptance Code Register is a bit pattern that an incoming message is compared to and must match before it will be accepted. An Acceptance Mask Register can be used to set certain bits to 'don't care' before the comparison process takes place. For the Ecan1000 a 'don't care' bit is any bit set to a one in a mask.

The first filter type 'SetSingleFilterStandard' allows an incoming message's 11 Bit ID to be compared, its RTR bit to be compared, and the first 2 bytes of the data to be compared. If all comparisons succeed then the message is accepted. The function takes a desired 11 Bit ACR and AMR for the ID comparison. Then a 1Bit RTR ACR and AMR. Finally a 16Bit AMR and ACR for comparison against the first 2 data bytes of the message. In the example the code looks like this:

First we make the ACR for the message ID equal to 0xFF;

```
Unsigned int ID_ACR = 0xFF;
```

Then we set the Acceptance mask register to say that we want all the bits of the ACR to be compared against the incoming ID. Note that a 1 bit means don't care and a zero bit means 'must match'.

```
Unsigned int ID_AMR = 0x0;
```

We don't care about the RTR bit so we set the Acceptance Mask register to a one.

```
Unsigned int RTR_AMR = 0x1;
```

Since we told the RTR AMR that we didn't care about the bit, it doesn't matter what we set the acceptance code register to.

```
Unsigned int RTR_ACR = 0x0;
```

We don't want to filter using the first two data bytes of the message so we set the AMR for the data bytes to 'Don't care' all bits.

```
Unsigned int DATA_AMR = 0xFFFF;
```

Since we set the data AMR bits to 'don't care' it doesn't matter what we set the ACR to.

```
Unsigned int DATA_ACR = 0x0;
```

Then we call the 'SetSingleFilterStandard' function to set up the filter.

```
Ecan_SetSingleFilterStandard(ReceiverHND, ID_ACR, ID_AMR, RTR_ACR,
RTR_AMR, DATA_ACR, DATA_AMR);
```

Now we have set up a filter that will accept all messages with an ID of 0xFF and we don't care what the RTR bit is set to or what the first two data bytes are.

The next Filter function 'SetSingleFilterExtended' sets up a filter for extended messages (those with a message ID of 29 bits). The filter allows all 29 bits of the ID to be compared as well as the RTR bit. If the compares succeed the message is accepted. The function takes a desired 29 bit ACR ID, a 29 bit AMR ID and a 1 bit RTR ACR and AMR. In the example, the code looks like this:

First we make the ACR for the Message ID equal to 0xFFFFF.

```
Unsigned long ID_ACR = 0xFFFFF;
```

Next we want to signal that we want all bits to match, so we set the AMR to 'must match'

```
Unsigned long ID_AMR = 0x0;
```

We don't care about the RTR bit so we set it to 'don't care'

```
Unsigned int RTR_AMR = 0x1;
```

Since we set the RTR Mask to be 'don't care' it doesn't matter what we set the code to.

```
Unsigned int RTR_ACR = 0x0;
```

Now we call the 'SetSingleFilterExtended' function to set up the filter.

```
Ecan_SetSingleFilterExtended(ReceiverHND, ID_ACR, ID_AMR, RTR_ACR,
RTR_AMR);
```

Now we will accept any extended message that has an ID of 0xFFFFF regardless of the state of the RTR bit.

The next filter function 'SetDualFilterStandard' sets up two filters for a standard message. In a two filter configuration the message must pass only one of the filters comparison tests in order to be accepted. In other words, the message will only be rejected if it fails both filter comparisons. The comparisons check all 11 bits of the message ID, the RTR bit and the first filter also checks the first 8 bits of data in the message. The example code looks like this:

First we will make the Acceptance Code Register for Filter 1 be 0x9.

```
Unsigned int ID_ACR1 = 0x9;
```

Now we want to make the Acceptance Mask Register say all bits 'must match'.

```
Unsigned int ID_AMR1 = 0x0;
```

Next we set up filter 2. Make the Acceptance Code Register for Filter 2 be 0x10.

```
Unsigned int ID_ACR2 = 0x10;
```

Now we make the AMR say that all bits 'must match'.

```
Unsigned int ID_AMR2 = 0x0;
```

We don't care about the state of the RTR bit for either filter so set the AMR for the RTR bit to 'don't care'.

```
Unsigned int RTR_AMR1 = 0x1;

Unsigned int RTR_AMR2 = 0x1;
```

Since we set the AMR's to 'don't care' it doesn't matter what we set the RTR's ACR to.

```
Unsigned int RTR_ACR1 = 0x0;

Unsigned int RTR_ACR2 = 0x0;
```

We also want to filter based on the first byte of data in the messages. We will accept messages with 0xF as the first data byte.

```
Unsigned int Data_ACR = 0xF;
```

Set the Data Acceptance Mask register to say that all bits 'must match'

```
Unsigned int Data_AMR = 0x0;
```

Now we send the filter information to 'SetDualFilterStandard'.

```
Ecan_SetDualFilterStandard(ReceiverHND, ID_ACR1, ID_AMR1,  ID_ACR2,
ID_AMR2, RTR_ACR1, RTR_AMR1, RTR_ACR2, RTR_AMR2, Data_ACR,
Data_AMR);
```

Now that the filters are set up we will receive all messages that have an ID of 0x9 AND the first data byte is 0xF OR if the message has a message ID of 0x10 it will pass the second filter test and be accepted.

The last function 'SetDualFilterExtended' allows a dual filter to be set up for extended messages (those messages with an ID length of 29 bits.). These filters allow bits 13 – 28 (the most significant 16 bits) of the message ID to be compared against each filter. The example code looks like the following:

Make the Acceptance Code for Filter 1 0xFFFF. (All 16 bits set).

```
Unsigned int ID_ACR1 = 0xFFFF;
```

Make the Acceptance Mask Register for filter 1 say 'all bits must match'.

```
Unsigned int ID_AMR1 = 0x0;
```

Make the Acceptance Code Register for filter 2 0xFFFE. (15 bits set).

```
Unsigned int ID_ACR2 = 0xFFFE;
```

Make the Acceptance Mask Register say 'All bits must match'.

```
Unsigned int ID_AMR2 = 0x0;
```

Now set up the filter using the 'SetDualFilterExtended' function

```
Ecan_SetDualFilterExtended(ReceiverHND, ID_ACR1, ID_AMR1, ID_ACR2,
ID_AMR2);
```

Now when an extended message that has the most significant 16 bits set (1FFFExxx) is found it will be accepted by filter 1. When a message with the most significant 15 bits set (1FFFCxxx) is found it will be accepted by filter 2.

API REFERENCE

Driver Initialization Functions

Ecan_CreateHandle

Syntax
HANDLE Ecan_CreateHandle(UCHAR DevNum=0, bool Ecan1000=false);

Description
This routine is used to open Ecan1000 or Ecan527 board. This function must be called before any API function call. At the end of the application program, the board must be closed with the CloseHandle function.

Parameters

DevNum:	Device number to load. Installed devices are numbered from 0 for each type (Ecan1000 and Ecan527).
Ecan1000:	Open Ecan1000 board – if TRUE, Ecan527 board – if FALSE.

Return Value

device handle	Board is opened successfully.
NULL	There was an error while closing the board.

To get extended error information, call GetLastError.

Example function call
…
HANDLE hDevice = Ecan_CreateHandle(0, true);   // Open the first of the Ecan1000 boards.
…
working with the board
…
CloseHandle(hDevice);   // Close the driver after work.

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following CreateFile function.

```
 sprintf( buff, "%s%d", LinkName, DevNum );
 HANDLE result = CreateFile(buff, GENERIC_READ |
GENERIC_WRITE, 0,  NULL, OPEN_EXISTING, 0,0);
```

Where LinkName is

CHAR *LinkName = "\\\\.\\Ecan527Device";

For the Ecan527 device and

CHAR *LinkName = "\\\\.\\Ecan1000Device";

For the Ecan1000 device

Ecan_GetBoardName

Syntax

ULONG Ecan_GetBoardName(HANDLE hDevice);

Description

This routine is used to determine the board name (Ecan1000 or Ecan527).

Parameters

hDevice:	Device handle.

Return Value

1000	For the Ecan1000 board.
527	For the Ecan527 board.

Example function call

```
…
if(Ecan_GetBoardName(hDevice )==527)
        MessageBox(NULL, “Ecan527 board”, “Ecan_GetBoardName”);
else
        MessageBox(NULL, “Ecan1000 board”, “Ecan_GetBoardName”);
…
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
 DeviceIoControl(hDevice, ECAN_IOCTL_GET_BOARD_NAME, NULL,
0, NULL, 0, &ret_code, NULL) ;
```

where ret_code is 527 or 1000 depending on board type.

Ecan_TestBoard

Syntax

bool Ecan_TestBoard(HANDLE hDevice);

Description

This routine is used to hardware test a board .

Parameters

hDevice:	Device handle.

Return Value

TRUE	Hardware test passed.
FALSE	Hardware test Failed.

Example function call

```
If(!Ecan_TestBoard(hDevice ))
{
        MessageBox(NULL, “Board test failed, check the hardware
settings.”, “Ecan_TestBoard”);
        CloseHandle(hDevice);
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_TEST,NULL, 0,NULL, 0,
&ret_code, NULL);
```

Where the return value of the DeviceIOControl call will be true if the board tested OK.

General Board Control Functions

Ecan_BusConfig

Syntax
bool Ecan_BusConfig(HANDLE hDevice, UCHAR BusTiming0, UCHAR BusTiming1, UCHAR ClockOut=0, UCHAR BusConfig=0xff);

Description
This routine sets Can Bit Timing and Bus configuration.
Ecan_StopBoard and Ecan_StartBoard should be called after this function has been used.

Parameters

hDevice:	Device handle.
BusTiming0	Value for the BUS TIMING REGISTER 0 (BIT TIMING REGISTER 0)
BusTiming1	Value for the BUS TIMING REGISTER 1 (BIT TIMING REGISTER 1)
ClockOut	Value that is used to set frequency divider at the external CLKOUT pin relative to the frequency of the external oscillator (XTAL). Divider value can be 1, 2, 4, 6, 8, 10, 12 or 14.
BusConfig	Value for the Output Control Register (Bus Configuration Register)

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call
```
…
Ecan_BusConfig( hDevice, 0, 0x14, 0, 0xff);    // 1Mbit/s
Ecan_StartBoard(hDevice);
```

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_SET_BUS_CONFIG, &msg,
sizeof(msg), NULL, 0, &ret_code, NULL);
```

Where msg is defined as:

```
typedef struct _ECAN_BUSCONFIG_STRUCTURE // Header of command
{
    UCHAR BusTiming0;
    UCHAR BusTiming1;
    UCHAR ClockOut;              // If=0 -not changing! 1=XTAL; 4=XTAL/4; 15=XTAL/15 etc.
    UCHAR BusConfig;       // For Ecan1000 = Output Control Register (if 0xff set default)
    _ECAN_BUSCONFIG_STRUCTURE()
    {
/* default bus timing values for
- bit-rate : 1 MBit/s
- oscillator frequency : 16 MHz, 0,1%
- maximum tolerated propagation delay : 623 ns
- minimum requested propagation delay : 23 ns */
            BusTiming0      = 0;     // baud rate prescaler : 1; SJW : 1
            BusTiming1      = 0x14;  // TSEG1 : 5; TSEG2 : 2
            ClockOut        = 0;
            BusConfig       = 0xff;
    };
} ECAN_BUSCONFIG_STRUCTURE, *PECAN_BUSCONFIG_STRUCTURE;
```

Ecan_SetupBoard

Syntax

bool Ecan_SetupBoard(HANDLE hDevice, HANDLE hEvent, OVERLAPPED *o_ptr=NULL, bool ReceiveIntEn=true, bool ErrorIntEn=false, bool TransmitIntEn=false, bool BusErrorIntEn=false, bool DataOverrunIntEn=false, bool ArbitrationLostIntEn=false, bool ErrorPassiveIntEn=false, bool WakeUpIntEn=false , ULONG RXSize = 32000, ULONG TXSize = 32000);

Description

This routine is used to setup the board.
Select interrupt sources for which interrupts the application will be notified.
Interrupts that are not selected will be handled inside the driver only. By default, the application gets Receive Interrupt only. This function must be called in RESET MODE only.

Parameters

HDevice:	Device handle.
HEvent:	Handle of the Event object used for interrupt handling.
o_ptr:	Pointer to the OVERLAPPED structure used to send Handle of the Event object to the driver.
ReceiveIntEn:	TRUE – turn on Receive Interrupt notification
ErrorIntEn:	TRUE – turn on Error Interrupt notification
TransmitIntEn:	TRUE – turn on Transmit Interrupt notification
BusErrorIntEn:	TRUE – turn on Bus Error Interrupt notification
DataOverrunIntEn:	TRUE – turn on Data Overrun Interrupt notification
ArbitrationLostIntEn:	TRUE – turn on Arbitration Lost Interrupt notification
ErrorPassiveIntEn:	TRUE – turn on Error Passive Interrupt notification
WakeUpIntEn:	TRUE – turn on Wake-Up Interrupt notification
RXSize:	Size of receive queue.
TXSize:	Size of transmit queue.

Return Value

TRUE	If there is no error.

FALSE If there is an error.
To get extended error information, call GetLastError.

Example function call

```
void far ReceiverCallBack()
{
        // Process the receiving interrupt, get an incoming message frame.
}
…
void main()
{
HANDLE hDevice = Ecan_CreateHandle(0, (true), 0x300, 7); // Open
the first of the Ecan1000 boards.
Ecan_SetupBoard (hDevice, ReceiverCallBack);
Ecan_StartBoard();
…
working with the board
…
Ecan_StopBoard();
CloseHandle(hDevice);    // Close the driver after work.
}
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_SETUP, &msg, sizeof(msg),
NULL, 0, &ret_code, o_ptr);
```

Where msg is an _ECAN_SETUP_STRUCTURE :

```
typedef struct _ECAN_SETUP_STRUCTURE // Header of command
{
        HANDLE          hEvent;
        bool    ReceiveIntEn;
        bool    ErrorIntEn;
        bool    TransmitIntEn;
        bool    BusErrorIntEn;
        bool    DataOverrunIntEn;
        bool    ArbitrationLostIntEn;
        bool    ErrorPassiveIntEn;
        bool        WakeUpIntEn;
        ULONG RX_QueueMaxSize;
        ULONG TX_QueueMaxSize;
        _ECAN_SETUP_STRUCTURE()
        {
                memset( this, 0, sizeof(*this)-(sizeof(ULONG)*2) );
                ReceiveIntEn = true;
                RX_QueueMaxSize = 32000; TX_QueueMaxSize = 32000;
        }
} ECAN_SETUP_STRUCTURE, *PECAN_SETUP_STRUCTURE;
```

Ecan_GetBuffer

Syntax

ULONG Ecan_GetBuffer(HANDLE hDevice, USHORT StartAddress, USHORT Count,
void* pBuffer, ULONG BuffSize)

Description

This routine allow direct read from the CAN controller's internal RAM (128-byte for Ecan1000 boards and 256-byte for Ecan527 boards). Through this function and Ecan_SetBuffer you can make access to all specific features of the board.

Parameters

hDevice:	Device handle.
StartAddress:	Offset from the start of the CAN controller's internal RAM
Count:	The number of bytes to be transferred from the board.
pBuffer:	Pointer to the user-supplied buffer that is to receive the data read from the controller.
BuffSize:	Buffer size.

Return Value

The number of bytes transferred to the buffer.

Example function call

```
UCHAR ModeReg;
Ecan_GetBuffer(hDevice, 0, 1, &ModeReg, sizeof(ModeReg)); // reads
Mode Register of Ecan1000
CString cStr;
cStr.Format("Mode Register = %d",  ModeReg);
MessageBox(NULL, cStr, " Ecan_GetBuffer");
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
```

```
DeviceIoControl(hDevice, ECAN_IOCTL_GET_BUFFER,
&header,sizeof(header), pBuffer, BuffSize, &ret_code, NULL);
```

Ecan_GetInterrupts

Syntax

UCHAR Ecan_GetInterrupts(HANDLE hDevice, ULONG *QueueSize=NULL, bool DontQueueUse=false)

Description

This routine is used to determine the interrupt source.

Parameters

hDevice:	Device handle.
QueueSize:	Pointer to a variable that receives the length of the receive queue. If DontQueueUse is TRUE, QueueSize receives 0.
DontQueueUse:	Normally an interrupt is received from the driver's interrupt queue. But if this parameter is TRUE, the interrupt is received directly from the hardware.

Return Value

The Interrupt Register value in format of the Ecan1000 board.

Remarks:

Interrupt register values:

Int_RI_Bit	0x01	receive interrupt bit
Int_TI_Bit	0x02	transmit interrupt bit
Int_EI_Bit	0x04	error warning interrupt bit
Int_DOI_Bit	0x08	data overrun interrupt bit
Int_WUI_Bit	0x10	wake-up interrupt bit
Int_EPI_Bit	0x20	error passive interrupt bit
Int_ALI_Bit	0x40	arbitration lost interrupt bit
Int_BEI_Bit	0x80	bus error interrupt bit

Example function call

```
// Process all interrupts from the queue
UCHAR Interrupt;
while((Interrupt = Ecan_GetInterrupts(hDevice))!=0) // While not all
accumulated
   //  interrupts are processed
{
// Process the interrupt (get received message for example).
}
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_GET_INTERRUPT_STATE,
NULL, 0, &str, sizeof(str), &ret_code, NULL) ;
```

Where str is an ECAN_INTERRUPT_STRUCTURE

```
typedef struct _ECAN_INTERRUPT_STRUCTURE // Header of command
{
    UCHAR  Interrupt;
    bool   DontQueueUse;
    USHORT        ReceiveQueueCount;      // For alignments on 4 bytes
    _ECAN_INTERRUPT_STRUCTURE(){Interrupt = 0; DontQueueUse = false;}
} ECAN_INTERRUPT_STRUCTURE, *PECAN_INTERRUPT_STRUCTURE;
```

Ecan_GetStatus

Syntax

bool Ecan_GetStatus(HANDLE hDevice,
PECAN_STATUS_STRUCTURE msg);

Description

This routine is used to receive the CAN controller status information.

Parameters

hDevice:	Device handle.
msg:	Pointer to the _ECAN_STATUS_STRUCTURE class object

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks:

ECAN_STATUS_STRUCTURE declaration:

```
typedef struct _ECAN_STATUS_STRUCTURE // Header of command
{
        bool BusOf;                  // BusOf status
        bool Warning;                // Erorr Warning
        bool WakeUp;                 // not use in Ecan1000
        bool TXOK;                   // in Ecan1000 - Transmission
Complete
        bool RXOK;                   // Receive Message Successfully (not
                                        used in Ecan1000)
        bool TS;                     // Transmit Status
        bool RS;                     // Receive Status
        bool TBS;                    // Transmit Buffer Status
        bool DOS;                    // Data Overrun Status
        bool RBS;                    // Receive Buffer Status
        UCHAR       Arbitration;     // Arbitration lost capture register value
        UCHAR       ErrorCode;       // Last Error Code register value
        bool DontQueueUse;           // Don’t use software queues in driver.
        UCHAR       Reserved[3];     // For alignments on 4 bytes
        _ECAN_STATUS_STRUCTURE(){memset( this, 0, sizeof(*this) );}
} ECAN_STATUS_STRUCTURE, *PECAN_STATUS_STRUCTURE;
```

Example function call

```
ECAN_STATUS_STRUCTURE Status;
if(Ecan_GetStatus( HANDLE hDevice, &Status))
{
if(Status.BusOf)
MessageBox (NULL, " Bus-of state!", " Ecan_GetStatus ");
}
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code
DeviceIoControl(hDevice, ECAN_IOCTL_GET_STATUS, NULL, 0, msg,
sizeof(*msg), &ret_code, NULL) ;
```

Ecan_LoadPortBitDir

Syntax
bool Ecan_LoadPortBitDir(HANDLE hDevice,
PECAN_PORTBITDIR_STRUCTURE msg);

Description
This function sets the direction of the pins on the DIO port (Ecan527 Only)

Parameters

hDevice:	Device handle.
PECAN_PORTBITDIR_STRUCTURE	
Bit7	True = Output, False = Input
Bit6	True = Output, False = Input
Bit5	True = Output, False = Input
Bit4	True = Output, False = Input
Bit3	True = Output, False = Input
Bit2	True = Output, False = Input
Bit1	True = Output, False = Input
Bit0	True = Output, False = Input

Return Value

True	Success.
False	Failure.

To get extended error information, call GetLastError.

Example Function Call
```
ECAN_PORTBITDIR_STRUCTURE PortDirMsg;
PortDirMsg.Bit7 = false;
PortDirMsg.Bit6 = false;
PortDirMsg.Bit5 = false;
PortDirMsg.Bit4 = false;
PortDirMsg.Bit3 = false;
PortDirMsg.Bit2 = false;
PortDirMsg.Bit1 = false;
PortDirMsg.Bit0 = false;
Ecan_LoadPortBitDir(hDevice,&PortDirMsg);
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ECAN_PORTBITDIR_STRUCTURE msg;
msg.Bit7 = false;
msg.Bit6 = false;
msg.Bit5 = false;
msg.Bit4 = false;
msg.Bit3 = false;
msg.Bit2 = false;
msg.Bit1 = false;
msg.Bit0 = false;

 ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_LOADPORTBITDIR,
&msg, sizeof(msg), NULL, 0, &ret_code, NULL)
```

Ecan_Read_Digital_IO

Syntax

bool Ecan_Read_Digital_IO(HANDLE hDevice,
PECAN_READWRITEDIGITALIO_STRUCTURE msg);

Description

This function reads a value from the digital IO port. Note: To get any value back from the port at least one pin must be set for input. See 'Ecan_LoadPortBitDir'.

Parameters

hDevice		Device handle
PECAN_READWRITEDIGITAL_IO		
	Value	Value read from the port.

Return Value

True	Success.
False	Failure.

To get extended error information, call GetLastError.

Example Function Call

```
ECAN_READWRITEDIGITALIO_STRUCTURE DIOValMsg;
Ecan_Read_Digital_IO(ReceiverHND,&DIOValMsg);
printf("Value Read %d  \n\r",DIOValMsg.Value);
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ECAN_READWRITEDIGITALIO_STRUCTURE msg;
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_READDIGITALIO, NULL,
0,&msg,sizeof(msg), &ret_code, NULL);
```

Ecan_SetBitRate

Syntax
bool Ecan_SetBitRate(HANDLE hDevice,BitRates BitRate)

Description
This function alters the communication bitrate. The board MUST be stopped and re-started for the change to take effect.

Parameters

hDevice		Device handle
BitRate		One of the BitRates enums:
	R1000000	1MB/s
	R800000	
	R500000	
	R400000	
	R250000	
	R200000	
	R160000	
	R125000	
	R100000	
	R80000	
	R62500	
	R50000	
	R40000	
	R31250	
	R25000	
	R20000	
	R16000	
	R15625	
	R12500	
	R10000	
	R8000	
	R7813	(actually 7812.5)
	R6150	
	R5000	

Return Value

True	Success.
False	Failure.

To get extended error information, call GetLastError.

Example Function Call
```
 Ecan_SetBitRate(ReceiverHND,R500000); // Set bitrate to 500K
```

Ecan_SetBuffer

Syntax

ULONG Ecan_SetBuffer(HANDLE hDevice, USHORT StartAddress, USHORT Count, void* Buffer, ULONG BuffSize)

Description

This routine allows direct write to the CAN controller's internal RAM (128-byte for Ecan1000 boards and 256-byte for Ecan527 boards). Through this function and Ecan_SetBuffer you can make access to all specific features of the board.

Parameters

hDevice:	Device handle.
StartAddress:	Offset from the start of the CAN controller's internal RAM
Count:	The number of bytes to be transferred from the buffer.
pBuffer:	A pointer to the user-supplied buffer that contains the data to be written to the controller.
BuffSize:	Buffer size.

Return Value

The number of bytes transferred to the controller.

Example function call

```
// Write 0 to the Mode Register of  the SJA1000 controller (Ecan1000
board)
UCHAR ModeReg=0;
Ecan_SetBuffer(hDevice, 0, 1, &ModeReg, sizeof(ModeReg));
Cstring cStr;
cStr.Format("Mode Register = %d",  ModeReg);
MessageBox (NULL, cStr, "Ecan_SetBuffer");
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_GET_BUFFER, &header,
sizeof(header), Buffer, BuffSize, &ret_code, NULL) ;
```

Where buffer is an ECAN_BUFFER_OPERATIONS structure:

```
typedef struct _ECAN_BUFFER_OPERATIONS // Header of command
{
      USHORT StartAddress;
      USHORT Count;
                              } ECAN_BUFFER_OPERATIONS, *PECAN_BUFFER_OPERATIONS;
```

Ecan_SetLeds

Syntax
bool Ecan_SetLeds(HANDLE hDevice, bool RedLed, bool GreenLed)

Description
This routine controls Leds on the Ecan527 - class board. . Note that pins 1 and 2 of the Digital IO are connected to the Led's. Turning the Led's on will output data on those pins. Also, the pins must be set to output (default) for the Led's to light (see Ecan_LoadPortBitDir).

Parameters

hDevice:	Device handle.
RedLed:	TRUE – enable red led.
	FALSE– disable red led.
RedGreen:	TRUE – enable green led.
	FALSE– disable green led.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call
Ecan_SetLeds(hDevice, false, true); // the Red led is off and the Green led is on.

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_SET_LEDS, &str, sizeof(str), NULL, 0, &ret_code, NULL) ;

Where str is an _ECAN_LEDS_STRUCTURE:

```
typedef struct _ECAN_LEDS_STRUCTURE
{
     bool      RedLed;
     bool      GreenLed;
     _ECAN_LEDS_STRUCTURE()
     {GreenLed=RedLed=false; }
}  ECAN_LEDS_STRUCTURE, *PECAN_LEDS_STRUCTURE;
```

Ecan_StartBoard

Syntax
bool Ecan_StartBoard(HANDLE hDevice);
Description
This routine switches board to OPERATING MODE.

Parameters

hDevice:	Device handle.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call

```
…
HANDLE hDevice = Ecan_CreateHandle(0, true, bVxd);      // Open the
first of the Ecan1000 boards.
…
working with the board
…
CloseHandle(hDevice);    // Close the driver after work.
```

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_START, NULL, 0, NULL, 0,
&ret_code, NULL);
```

Where the return value from the DeviceIOControl function indicates success or failure.

Ecan_StopBoard

Syntax
bool Ecan_StartBoard(HANDLE hDevice);

Description
This routine switches board to RESET MODE and stops all operations.

Parameters

hDevice:	Device handle.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call
…
HANDLE hDevice = Ecan_CreateHandle(0, true, bVxd); // Open the first of the Ecan1000 boards.
…
working with the board
…
CloseHandle(hDevice); // Close the driver after work.

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_STOP, NULL, 0, NULL, 0,
&ret_code, NULL);
```

Where the return value of the DeviceIOControl function indicates success or failure.

Ecan_Write_Digital_IO

Syntax

```
bool Ecan_Write_Digital_IO( HANDLE hDevice,
PECAN_READWRITEDIGITALIO_STRUCTURE msg );
```

Description

This routine writes a value out to the DIO port. Note: For any value to appear on the port at least one of the pins must be set to output using 'Ecan_LoadPortBitDir'.

Parameters

hDevice	Device handle
PECAN_READWRITEDIGITAL_IO	
Value	Value to write to the port

Return Value

True	Success.
False	Failure.

To get extended error information, call GetLastError.

Example Function Call

```
ECAN_READWRITEDIGITALIO_STRUCTURE DIOValMsg;
DIOValMsg.Value = VALUEOUT;
Ecan_Write_Digital_IO(ReceiverHND,&DIOValMsg);
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ECAN_READWRITEDIGITALIO_STRUCTURE msg;
msg.Value = Value;
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_WRITEDIGITALIO, &msg,
sizeof(msg), NULL, 0, &ret_code, NULL) ;
```

Message Manipulation Functions

Ecan_GetMessage

Syntax
bool Ecan_GetMessage(HANDLE hDevice, PECAN_MESSAGE_STRUCTURE msg);

Description
This routine is used to retrieve a received message.

Parameters

hDevice:	Device handle.
msg:	A pointer to the ECAN_MESSAGE_STRUCTURE.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks
ECAN_MESSAGE_STRUCTURE declaration:

```
typedef struct _ECAN_MESSAGE_STRUCTURE // Header of command
{
        UCHAR       Channel;      // Not used in the Ecan1000.
                                  //In Ecan527, Channel 0 - Default
                                  Transmitting or Default Receiving
                                  Message Object
                                  //1 - 15 Message Object by number.
        bool   Extended;          // If true - Extended frame format
        bool   Remote;            // if true - remote frame (Attention For
                                  Ecan527! this value = inverse direction
                                  bit)
        UCHAR       ID_0;         // Identifier 0 (Arbitration 0 - in Ecan527
                                  documentation)
        UCHAR       ID_1;         // Identifier 1 (Arbitration 1 - in Ecan527
                                  documentation)
        UCHAR       ID_2;         // Identifier 2 (Arbitration 2 - in Ecan527
                                  documentation)
        UCHAR       ID_3;         // Identifier 3 (Arbitration 3 - in Ecan527
                                  documentation)
        UCHAR       DataLength;   // Data Length Code (DLC)
        UCHAR        Data[8];         // Data Bytes
        bool    NextMsg;              // if true – There is another message
available in the queue
        bool    DontQueueUse;         // Don’t use driver’s software queues
        UCHAR        Reserved[2]; // For alignments on 4 bytes
        _ECAN_MESSAGE_STRUCTURE(){ Clear(); };
        void Clear(){ memset( this, 0, sizeof(*this) ); }
        // Identifiers services functions
        ULONG GetID()
        {
                ULONG ret = ID_0; ret = (ret<<8)|ID_1;
                if( Extended )ret = (((ret<<8)|ID_2)<<8)|ID_3;
                return ret;
        };
        void SetID(ULONG ID)
        {
                if( Extended ){ ID_3 = (UCHAR)ID; ID = ID>>8; ID_2 =
(UCHAR)ID; ID = ID>>8; };
                ID_1 = (UCHAR)ID; ID = ID>>8; ID_0 = (UCHAR)ID;
        };

} ECAN_MESSAGE_STRUCTURE,
*PECAN_MESSAGE_STRUCTURE;
```

Example function call

```
ECAN_MESSAGE_STRUCTURE Msg;//
ECAN_MESSAGE_STRUCTURE with default settings
if (!Ecan_GetMessage(hDrvice, &Msg ) )
{
        MessageBox (NULL, "Can not receive a message!", “
Ecan_GetMessage”);
}
else
MessageBox (NULL, "Message accepted!", “ Ecan_GetMessage”);
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_GET_MESSAGE, NULL, 0,
msg, sizeof(*msg), &ret_code, NULL) ;
```

Where the return value of the DeviceIOControl function indicates success or failure.

Ecan_GetQueuesCounts

Syntax

```
bool Ecan_GetQueuesCounts( HANDLE hDevice, ULONG *TX_Count,
ULONG *RX_Count, bool ClearRX=false, bool ClearTX=false );
```

Description

This routine is used to receive the messages queues length and/or to clear these queues.

Parameters

hDevice:	Device handle.
TX_Count:	Pointer to a variable that receives the length of the transmit queue.
RX_Count:	Pointer to a variable that receives the length of the receive queue.
ClearRX:	TRUE – clears the receive queue.
ClearTX:	TRUE – clears the transmit queue.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call

```
// Wait until the transmit queue will freed
ULONG TX_Count = 0x80;
while ( TX_Count >= 1 )
{
Ecan_GetQueuesCounts( hTransmitter, &TX_Count);
}
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_GET_QUEUES_COUNTS,
NULL, 0, &msg, sizeof(msg), &ret_code, NULL);
```

Where msg is an ECAN_QUEUES_COUNTS_STRUCTURE

```
typedef struct _ECAN_QUEUES_COUNTS_STRUCTURE // Header of command
{
ULONG    RX_Count;
ULONG    TX_Count;
bool        ClearRX;
bool        ClearTX;
UCHAR Reserved[2];          // For alignments on 4 bytes
_ECAN_QUEUES_COUNTS_STRUCTURE()            { memset( this, 0, sizeof(*this) ); };
} ECAN_QUEUES_COUNTS_STRUCTURE, *PECAN_QUEUES_COUNTS_STRUCTURE;
```

Ecan_SendCommand

Syntax

bool Ecan_SendCommand(HANDLE hDevice, bool TR=false, bool RRB=false, bool AT=false, bool CDO=false, bool SRR=false);

Description

A command bit initiates an action within the transfer layer of the SJA1000 (Ecan1000 board).

Parameters

hDevice:	Device handle.
TR:	Transmission Request.
RRB:	Release Receive Buffer.
AT:	Abort Transmission.
CDO:	Clear Data Overrun.
SRR:	Self Reception Request.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call

Ecan_SendCommand(hDevice, false, false, true); // Abort current Transmission

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_SEND_COMMAND, &msg,
sizeof(msg), NULL, 0, &ret_code, NULL) ;
```

Where msg is an ECAN_COMMAND_STRUCTURE

```
typedef struct _ECAN_COMMAND_STRUCTURE
{
        bool    TR;                          // Transmission Request
        bool    RRB;                         // Release Receive Buffer
        bool    AT;                          // Abort Transmission
```

```
    bool    CDO;                        // Clear Data Overrun
    bool    SRR;                        // Self Reception Request
    UCHAR Reserved[3];                  // For alignments on 4 bytes
    _ECAN_COMMAND_STRUCTURE(){ Clear(); };
    void Clear(){ memset( this, 0, sizeof(*this) ); }
}ECAN_COMMAND_STRUCTURE, *PECAN_COMMAND_STRUCTURE;
```

Ecan_SendMessage

Syntax

```
bool Ecan_SendMessage( HANDLE hDevice,
PECAN_MESSAGE_STRUCTURE msg);
```

Description

This routine is used to transmit message.

Parameters

hDevice:	Device handle.
msg:	A pointer to the ECAN_MESSAGE_STRUCTURE (see Ecan_GetMessage for details).

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call

```
ECAN_MESSAGE_STRUCTURE Msg;
Msg.Extended=true;        // Extended frame format
Msg.Channel=0;            // by default
Msg.DataLength=2;         // 2 bytes to transmit
Msg.Data[0]=0xa5;         // Data 0
Msg.Data[1]=0x5a;         // Data 1
Msg.SetID(0x01);          // Message Identifier

// Send message through the transmit queue (and the default transmit
object in Ecan527).
Ecan_SendMessage( hDevice, &Msg );
…
```

Optional Calling Syntax

For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_SEND_MESSAGE, msg,
sizeof(*msg), NULL, 0, &ret_code, NULL) ;
```

Ecan_SetDualFilterExtended

Syntax

bool Ecan_SetDualFilterExtended(HANDLE hDevice,unsigned int ID_ACR1,unsigned int ID_AMR1,unsigned int ID_ACR2,unsigned int ID_AMR2)

Description

This function sets up dual filters for extended messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR1	16 Bit Acceptance Code for the message ID for Filter 1
ID_AMR1	16 Bit Acceptance Mask for the message ID for Filter 1
ID_ACR2	16 Bit Acceptance Code for the message ID for Filter 2
ID_AMR2	16 Bit Acceptance Mask for the message ID for Filter 2

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Ecan_SetDualFilterStandard

Syntax

```
bool Ecan_SetDualFilterStandard(HANDLE hDevice,unsigned int
ID_ACR1,unsigned int ID_AMR1,unsigned int ID_ACR2,unsigned int ID_AMR2,
unsigned int RTR_ACR1,unsigned int RTR_AMR1,unsigned int
RTR_ACR2,unsigned int RTR_AMR2, unsigned int Data_ACR,unsigned int
Data_AMR)
```

Description

This function sets up dual filters for standard messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR1	11 Bit Acceptance Code for the message ID for Filter 1
ID_AMR1	11 Bit Acceptance Mask for the message ID for Filter 1
ID_ACR2	11 Bit Acceptance Code for the message ID for Filter 2
ID_AMR2	11 Bit Acceptance Mask for the message ID for Filter 2
RTR_ACR1	1 Bit Acceptance Code for the RTR Bit for Filter 1
RTR_AMR1	1 Bit Acceptance Mask for the RTR Bit for Filter 1
RTR_ACR2	1 Bit Acceptance Code for the RTR Bit for Filter 2
RTR_AMR2	1 Bit Acceptance Mask for the RTR Bit for Filter 2
Data_ACR	8 Bit Acceptance Code for the first data byte for Filter 1
Data_AMR	8 Bit Acceptance Mask for the first data byte for Filter 1

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Ecan_SetFilter

Syntax
bool Ecan_SetFilter(HANDLE hDevice, PECAN_FILTER_STRUCTURE msg);

Description
This routine sets message filters and masks configurations.

Parameters

hDevice:	Device handle.
msg:	Pointer to the _ECAN_FILTER_STRUCTURE class object, containing information about filters and masks to be set.

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks:
ECAN_STATUS_STRUCTURE declaration:

```
typedef struct _ECAN_FILTER_STRUCTURE // Header of command
{
        UCHAR AC0;    // Acceptance Code Register 0 (For Ecan527 - Not used)
        UCHAR AC1;    // Acceptance Code Register 1 (For Ecan527 - Not used)
        UCHAR AC2;    // Acceptance Code Register 2 (For Ecan527 - Not used)
        UCHAR AC3;    // Acceptance Code Register 3 (For Ecan527 - Not used)
        UCHAR AM0;    // Acceptance Mask Register 0 (For Ecan527 - Global Mask Standart (Address-06H))
        UCHAR AM1;    // Acceptance Mask Register 1 (For Ecan527 - Global Mask Standart (Address-07H))
        UCHAR AM2;    // Acceptance Mask Register 2 (For Ecan527 - Not used)
        UCHAR AM3;    // Acceptance Mask Register 3 (For Ecan527 - Not used)
                      // Not used in the Ecan1000
        UCHAR ME0;    // Global Mask Extended (Address-08H)
        UCHAR ME1;    // Global Mask Extended (Address-09H)
        UCHAR ME2;    // Global Mask Extended (Address-0AH)
        UCHAR ME3;    // Global Mask Extended (Address-0BH)
        UCHAR MM0;    // Message 15 mask (Address 0CH)
        UCHAR MM1;    // Message 15 mask (Address 0DH)
        UCHAR MM2;    // Message 15 mask (Address 0EH)
        UCHAR MM3;    // Message 15 mask (Address 0FH)

        bool    DualFilter; // For Ecan1000 only
        bool    DontQueueUse;
        UCHAR Reserved[2];      // For alignments on 4 bytes
        _ECAN_FILTER_STRUCTURE(){ Clear(); };
        void Clear()
```

```
    void SetAcceptCode(ULONG ID, bool LongFormat=true)

    ULONG GetAcceptCode(bool LongFormat=true)

    void SetAcceptMask(ULONG ID, bool LongFormat=true, bool Ecan527=false)

    ULONG GetAcceptMask(bool LongFormat=true, bool Ecan527=false)

    // For Ecan 527
    void SetExtended(ULONG ID)

    ULONG GetExtended()

    void SetMessageMask(ULONG ID)

    ULONG GetMessageMask()


}ECAN_FILTER_STRUCTURE, *PECAN_FILTER_STRUCTURE;
```

Example function call

```
ECAN_FILTER_STRUCTURE Filter;
…
// Fill the filter structure.
…
bool Ecan_SetFilter(hDevice, Filter);
```

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_SET_FILTER, msg,
sizeof(*msg), NULL, 0, &ret_code, NULL) ;
```

Ecan_SetSingleFilterExtended

Syntax

bool Ecan_SetSingleFilterExtended(HANDLE hDevice,unsigned long ID_ACR,unsigned long ID_AMR,unsigned int RTR_ACR,unsigned int RTR_AMR)

Description

This function sets up a single filter for extended messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR	29 Bit Acceptance Code for the message ID
ID_AMR	29 Bit Acceptance Mask for the message ID
RTR_ACR	1 Bit Acceptance Code for the RTR Bit
RTR_AMR	1 Bit Acceptance Mask for the RTR Bit

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Ecan_SetSingleFilterStandard

Syntax

bool Ecan_SetSingleFilterStandard(HANDLE hDevice,unsigned int ID_ACR,unsigned int ID_AMR,unsigned int RTR_ACR,unsigned int RTR_AMR,unsigned int Data_ACR,unsigned int Data_AMR)

Description

This function sets up a single filter for standard messages on the Ecan1000 card.

Parameters

HDevice	Device Handle
ID_ACR	11 Bit Acceptance Code for the message ID
ID_AMR	11 Bit Acceptance Mask for the message ID
RTR_ACR	1 Bit Acceptance Code for the RTR Bit
RTR_AMR	1 Bit Acceptance Mask for the RTR Bit
DATA_ACR	16Bit Acceptance Code for the first 2 data bytes
DATA_AMR	16Bit Acceptance Mask for the first 2 data bytes

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks

See the section titled ‘Message Filtering’ in this manual for more info.

Ecan_MessageObjectSetup

Syntax
bool Ecan_MessageObjectSetup(HANDLE hDevice, PECAN_MESSAGE_OBJECT_SETUP_STRUCTURE msg);

Description
This routine allows the application to setup message objects

Parameters

hDevice:	Device handle.
msg:	Handle to the _ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE

Return Value

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Remarks:
_ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE declaration

```
typedef struct _ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE // Header of command
{
        UCHAR Channel;// Number of Message Object
                                // if Channel==0, then using default Transmit or Receive Message Object
        MESSAGE_OBJECT_STATE State; // Program object for one of three tasks
                                // MO_TRANSMIT-transmit
                                // MO_REMOTE_TRANSMIT - transmit after receiving remote frame whith same ID
                                // MO_RECEIVE - receive

        bool    Valid;          // If true  - enable Message Object after setup
        bool    Extended;       // If true - use Extended frame format
        bool    RXIE;           // Enable Receive Message Interrupt for this object
        bool    TXIE;           // Enable Transmit Message Interrupt for this object
        bool    DontQueueUse;// if FALSE for MO_TRANSMIT, then changes Default Transmitting Object to this
        bool    MakeDefault;
        UCHAR ID_0;             // Identifier 0 (Arbitration 0 - in Ecan527 documentation)
        UCHAR ID_1;             // Identifier 1 (Arbitration 1 - in Ecan527 documentation)
        UCHAR ID_2;             // Identifier 2 (Arbitration 2 - in Ecan527 documentation)
        UCHAR ID_3;             // Identifier 3 (Arbitration 3 - in Ecan527 documentation)
        UCHAR   Reserved[3];
        _ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE()
        { Clear(); }
        void Clear(){ memset( this, 0, sizeof(*this) ); }
        // Identifiers services functions
        ULONG GetID()
        {
                ULONG ret = ID_0; ret = (ret<<8)|ID_1;
```

```
            if( Extended )ret = (((ret<<8)|ID_2)<<8)|ID_3;
            return ret;
    };
    void SetID(ULONG ID)
    {
            if( Extended ){ ID_3 = (UCHAR)ID; ID = ID>>8;      ID_2 = (UCHAR)ID; ID = ID>>8; };
            ID_1 = (UCHAR)ID; ID = ID>>8; ID_0 = (UCHAR)ID;
    };

} ECAN_MESSAGE_OBJECT_SETUP_STRUCTURE, *PECAN_MESSAGE_OBJECT_SETUP_STRUCTURE;
```

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice, ECAN_IOCTL_MESSAGE_OBJECT_SETUP,
msg, sizeof(*msg), NULL, 0, &ret_code, NULL) ;
```

Ecan_AllowBufferOverwrite

Description
Calling this function prevents the driver from returning an Interrupt 0xff and resetting the card when the receive queue is full. Instead, the oldest data in the queue is discarded and overwritten by the newest data and An Interrrupt 0xff is returned.

Syntax
bool Ecan_AllowBufferOverwrite(HANDLE hDevice);

Parameters

hDevice:	Device handle.

Return Values

TRUE	If there is no error.
FALSE	If there is an error.

To get extended error information, call GetLastError.

Example function call

```
// Tell driver to allow buffer overwrite
Ecan_AllowBufferOverwrite(ReceiverHND);
```

Optional Calling Syntax
For those who wish to communicate directly with the driver, rather than using the provided DLL, this call uses the following DeviceIOControl function.

```
ULONG ret_code;
DeviceIoControl(hDevice,
ECAN_IOCTL_ALLOW_MESSAGE_OVERWRITE, NULL,0, NULL, 0,
&ret_code, NULL) ;
```

LIMITED WARRANTY

RTD Embedded Technologies, Inc. warrants the hardware and software products it manufactures and produces to be free from defects in materials and workmanship for one year following the date of shipment from RTD Embedded Technologies, INC. This warranty is limited to the original purchaser of product and is not transferable.

During the one year warranty period, RTD Embedded Technologies will repair or replace, at its option, any defective products or parts at no additional charge, provided that the product is returned, shipping prepaid, to RTD Embedded Technologies. All replaced parts and products become the property of RTD Embedded Technologies. Before returning any product for repair, customers are required to contact the factory for an RMA number.

THIS LIMITED WARRANTY DOES NOT EXTEND TO ANY PRODUCTS WHICH HAVE BEEN DAMAGED AS A RESULT OF ACCIDENT, MISUSE, ABUSE (such as: use of incorrect input voltages, improper or insufficient ventilation, failure to follow the operating instructions that are provided by RTD Embedded Technologies, "acts of God" or other contingencies beyond the control of RTD Embedded Technologies), OR AS A RESULT OF SERVICE OR MODIFICATION BY ANYONE OTHER THAN RTD Embedded Technologies. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES ARE EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND RTD Embedded Technologies EXPRESSLY DIS-CLAIMS ALL WARRANTIES NOT STATED HEREIN. ALL IMPLIED WARRANTIES, INCLUDING IMPLIED WARRANTIES FOR MECHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED TO THE DURATION OF THIS WARRANTY. IN THE EVENT THE PRODUCT IS NOT FREE FROM DEFECTS AS WARRANTED ABOVE, THE PURCHASER'S SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. UNDER NO CIRCUMSTANCES WILL RTD Embedded Technologies BE LIABLE TO THE PURCHASER OR ANY USER FOR ANY DAMAGES, INCLUDING ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, EXPENSES, LOST PROFITS, LOST SAVINGS, OR OTHER DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES FOR CONSUMER PRODUCTS, AND SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

RTD Embedded Technologies, Inc.

103 Innovation Blvd.

State College PA 16803-0906

USA

Our website: www.rtd.com